Fully Automatic Parallelization

- There are huge amounts of source code which is sequential.
- Using OpenMP is semi-automatic
- For the last 50 years or so, there has been a quest for automatically parallelizing sequential programs.
- An approach to parallelize source code
  - First try a parallelizing compiler and see what happens
  - If it fails then look for compiler feedback and see if you can modify the source
  - If not useful, try OpenMP
  - If not useful, parallelize manually

Safety of Parallelization

- Does the parallel program produce the same output?
- Invalid if data-races are created, obviously.
- When a for-loop is parallelized, the iterations are run in an unpredictable order.
- Note: changing the iteration order can cause numerical problems
- Note above applies also to sequential programs.

From Simple to Hard Parallelization Problems

- Easiest case: loops with matrix computations and with known loop bounds and array indexes that are linear functions of the loop variables
- We will be more precise shortly
- Very complicated case: code with dynamically allocated data structures with many pointers
- It would be very hard to automatically parallelize Lab 0
- This lecture focuses on matrix computations

Inner vs Outer Loop Parallelization

- In the course EDAN75 Optimizing Compilers you can learn about inner loop parallelization which is used e.g. for automatic SIMD vectorization and software pipelining.
- Here the focus instead is on automatic parallelization for multicores, i.e. outer loop parallelization.
- The foundations for inner and outer loop parallelization are similar, since they both rely on data dependence analysis.

True data dependences

- A **true dependence**:

```
S1: x = a + b;
S2: y = x + 1;
```

- It is written $S_1 \delta^t S_2$.
- S_1 must execute before S_2 in any transformed program.

Data Dependences at Different Levels

- Data dependences can be at several different levels:
  - Instructions
  - Statements
  - Loop iterations
  - Functions
  - Threads
- Parallelizing compilers usually find parallelism between different loop iterations of a loop.
- If the compiler can determine that there are no dependences between loop iterations then it can either:
  - Produce parallel machine code, or
  - Produce source code with OpenMP `#pragma parallel for` directives.
- If there are dependences, it may still be possible to execute the loop in parallel since perhaps the loop iterations are not totally ordered.

Total vs Partial Order and Loop Iterations

- Integers are totally ordered since we can determine which of a and b is greater if $a \neq b$.
- Consider a directed acyclic graph. In topological sorting you can process any node u if all predecessors of u already have been processed.
- Obviously, we should not execute a loop iteration before its input data has been computed.
- In executing a loop in parallel we perform a topological sort of the loop iterations.
- Conceptually, topological sorting is the major work in parallelization.
- No topological search is performed during compilation or runtime to determine which iterations can be executed, though.
- Instead, new loops are *computed* (i.e. created) by the compiler.
- If the iterations are a total order no parallelization can be done

Three more data dependences

- In an **anti dependence**, written $I_1 \delta^a I_2$, I_1 reads a memory location later overwritten by I_2.
- In an **output dependence**, written $I_1 \delta^o I_2$, I_1 writes a memory location later overwritten by I_2.
- In an **input dependence**, written $I_1 \delta^i I_2$, both I_1 and I_2 read the same memory location.
- The first three types of dependences create partial orderings among all iterations, which parallelizing compilers exploit by ordering iterations to improve performance.
- Input dependences can give a hint to the compiler that some data will be used so it can try to keep it in the cache (by reordering iterations in a suitable way).

Loop Level Data Dependences

- In the loop

```
for (i = 3; i < 100; i += 1)
        a[i] = a[i-3] + x;
```

- There is a true dependence from iteration i to iteration $i+3$.
- Iteration $i = 3$ writes to a_3 which is read in iteration $i = 6$.
- A loop level true dependence means one iteration writes to a memory location which a later reads.

Perfect Loop Nests

- A **perfect loop nest** L is a nest of m nested **for** loops $L_1, L_2, ...L_m$ such that the body of $L_i, i < m$, consists of L_{i+1} and the body of L_m consists of a sequence of assignment statements.
- For $1 < r \leq m$ p_r and q_r are linear functions of $I_1, ..., I_{r-1}$.

for $(I_1 = p_1;\ I_1 <= q_1;\ I_1 + = 1)$ {
    for $(I_2 = p_2;\ I_2 <= q_2;\ I_2 + = 1)$ {
        $\vdots$
        for $(I_m = p_m;\ I_m <= q_m;\ I_m + = 1)$ {
            $h(I_1, I_2, ..., I_m)$;
        }
    }
}

Example Perfect Loop Nest

- All assignments, **except** to the loop index variables are in the innermost loop.
- There may be any number of assignment statements in the innermost loop.

```
for (i = 0; i < 100; i += 1) {
        for (j = 3 + i; j <  2 * i + 10; j += 1) {
                for (k = i - j; k < j - i; k += 1) {
                        a[i][j][k] += b[k][j][i];
                }
        }
}
```

Loop Bounds

- The lower bound for l_1 is $p_{10} \leq l_1$.
- The lower bound for l_2 is

$$\begin{aligned} l_2 &\geq p_{20} + p_{21} l_1 \\ p_{20} &\leq l_2 - p_{21} l_1 \\ p_{20} &\leq -p_{21} l_1 + l_2 \end{aligned}$$

- The lower bound for l_3 is

$$\begin{aligned} l_3 &\geq p_{30} + p_{31} l_1 + p_{32} l_2 \\ p_{30} &\leq l_3 - p_{31} l_1 - p_{32} l_2 \\ p_{30} &\leq -p_{31} l_1 - p_{32} l_2 + l_3 \end{aligned}$$

and so forth. We represent this on matrix form as $\mathrm{p}_0 \leq \mathrm{IP}$, or... see next slide.

Loop Bounds on Matrix Form

- $\mathrm{P} = \begin{pmatrix} 1 & -p_{21} & -p_{31} & \dots & -p_{m1} \\ 0 & 1 & -p_{32} & \dots & -p_{m2} \\ 0 & 0 & 1 & \dots & -p_{m3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$ and $\mathrm{p}_0 = (p_{10}, p_{20}, \dots, p_{m0})$.
- Similarly, the upper bounds are represented as $\mathrm{IQ} \leq \mathrm{q}_0$.
- The loop bounds, thus, are represented by the system:

$$\left.\begin{array}{ccccc} \mathrm{p}_0 & \leq & \mathrm{IP} & & \\ & & \mathrm{IQ} & \leq & \mathrm{q}_0 \end{array}\right\}$$

Example Non-Perfect Loop Nest

- The assignment to c_{ij} before the innermost loop makes it a non-perfect loop nest.
- Sometimes non-perfect loop nest can be split up, or **distributed** into perfect loop nests.
- See next slide.

```
for (i = 0; i < 100; i += 1) {
        for (j = 0; j < 100; j += 1) {
                c[i][j] = 0;
                for (k = 0; k < 100; k += 1) {
                        c[i][j] += a[i][k] * b[k][j];
                }
        }
}
```

Loop Distribution

- Result of loop distribution.

```
for (i = 0; i < 100; i += 1)
        for (j = 0; j < 100; j += 1)
                c[i][j] = 0;
for (i = 0; i < 100; i += 1)
        for (j = 0; j < 100; j += 1)
                for (k = 0; k < 100; k += 1)
                        c[i][j] += a[i][k] * b[k][j];
```

Some Terminology

- The index vector $\mathbf{I} = (I_1, I_2, ..., I_m)$ is a vector with index variables.
- The index values of $\mathbf{L}$ are the values of $(I_1, I_2, ..., I_m)$.
- The index space of $\mathbf{L}$ is the subspace of Z^m consisting of all the index values.
- An **affine array reference** is an array reference in which all subscripts are linear functions of the loop index variables.

Easy non-affine references

- Data dependence analysis is normally restricted to affine array references.
- In practice, however, subscripts often contain **symbolic constants** as shown below which is test `s171` in the C version of the Argonne Test Suite for Vectorising Compilers.
- There is no dependence between the iterations in this test.

```
for (i=0; i<n; i++)
    a[i*n] = a[i*n] + b[i];
```

Problematic Non-Affine Index Functions

- In the loop

```
scanf("%d", &x);

for (i = 3; i < 100; i += 1) {
S1:     a[i]   = a[x] + 1;
S2:     b[i]   = b[c[i-1]] + 2;
S3:     d[i]   = d[2 * i * i * i - 3 * i * i ] + 3;
}
```

- Some compilers do runtime testing to take care of S_1 but it may cause too much overhead if many variables must be checked.

Representing Array References

- Let X be an n-dimensional array. Then an affine reference has the form:
- $X[a_{11}i_1 + a_{21}i_2 ... a_{m1}i_m + a_{01}]...[a_{1n}i_1 + a_{2n}i_2 ... a_{mn}i_m + a_{0n}]$
- This is conveniently represented as a matrix and a vector $X[\mathrm{IA} + \mathrm{a}_0]$, where
- $\mathrm{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$ and $\mathrm{a}_0 = (a_{10}, a_{20}, ..., a_{n0})$.
- We will refer to A and a_0 as the **coefficient matrix** and the **constant term**, respectively.

An Example

```
for (i = 0; i < 100; i += 1)
        for (j = 2*i + 4; j < i + 40; j += 1)
                a[2i-3j-1][2i+j-3] = f(a[-3i+4j+1][-i+2j+7]);
```

- The above loop nest has the following two array reference representations:
  $A = \begin{pmatrix} 2 & 2 \\ -3 & 1 \end{pmatrix}$ and $a_0 = (-1, -3)$.
  $B = \begin{pmatrix} -3 & -1 \\ 4 & 2 \end{pmatrix}$ and $b_0 = (1, 7)$.

The Data Dependence Equation

- For two references $X[\mathrm{IA} + \mathrm{a}_0]$ and $X[\mathrm{IB} + \mathrm{b}_0]$ to refer to the same array element there must be two index values, i and j such that $\mathrm{iA} + \mathrm{a}_0 = \mathrm{jB} + \mathrm{b}_0$ which we can write as $\mathrm{iA} - \mathrm{jB} = \mathrm{b}_0 - \mathrm{a}_0$.
- This system of Diophantine equations has n (the dimension of the array X) scalar equations and $2m$ variables, where m is the nesting depth of the loop.
- It can also be written in the following form:

$$(\mathrm{i};\mathrm{j})\begin{pmatrix} \mathrm{A} \\ -\mathrm{B} \end{pmatrix} = \mathrm{b}_0 - \mathrm{a}_0.$$

- We solve the system of linear Diophantine equations above using a method presented shortly.

Dependence Distances

- Let $\prec_\ell$ be a relation in Z^m such that $\mathrm{i} \prec \mathrm{j}$ if $i_1 = j_1$, $i_2 = j_2$, ..., $i_{l-1} = j_{l-1}$, and $i_l < j_l$.
- For example: $(1, 3, 4) \prec_3 (1, 3, 9)$.
- The lexicographic order $\prec$ in Z^m is the union of all the relations $\prec_\ell$: $\mathrm{i} \prec \mathrm{j}$ iff $\mathrm{i} \prec_\ell \mathrm{j}$ for some ℓ in $1 \leq \ell \leq m$.
- The sequential execution of the iterations of a loop nest follows the lexicographic order.
- Assume that $(\mathrm{i}; \mathrm{j})$ is a solution and that $\mathrm{i} \prec \mathrm{j}$. Then $\mathrm{d} = \mathrm{j} - \mathrm{i}$ is the **dependence distance** of the dependence.

Uniform Dependence Distance

- If a dependence distance d is a constant vector then the dependence is said to be uniform.
- Examples:
  - d $= (1, 2)$ is uniform — required for parallelization.
  - d $= (1, t_2)$ is nonuniform — loop cannot be parallelized.
- All unique d are put in a matrix as rows — but row order does not matter since it is really just a set of all d

Loop Independent and Loop Carried Dependences

- A loop independent dependence is a dependence such that $d = j - i = (0, ..., 0)$.
- A loop independent dependence does not prevent concurrent execution of different iterations of a loop. Rather, it constrains the scheduling of instructions in the loop body.
- A loop carried dependence is a dependence which is not loop independent, or, in other words, the dependence is between two different iterations of a loop nest.
- A dependence has level ℓ if in $d = j - i$, $d_1 = 0, d_2 = 0, ..., d_{l-1} = 0$, and $d_l > 0$.
- Only a loop carried dependence has a level, and it is only the loop at that level which needs to be executed sequentially.

The GCD Test

- Recall that a Diophantine equation $ax + by = c$ has a solution only if $gcd(a, b)$ divides c
- The GCD test was invented at Texas Instruments and first described 1973.
- Consider the loop

```
for (i = lb; i <= ub; ++i)
        x[ a1 * i + c1] = x[a2 * i + c2] + y;
```

- To prove independence, we must show that the Diophantine equation

$$a_1 i_1 - a_2 i_2 = c_2 - c_1$$

has no solutions.

- ```
  for (i = 1; i <= 100; ++i)
          x[ 2 i ] = x[2 * i + 1] + y; // even and odd
  ```
  ```

Weaknesses of The GCD Test

- There are two weaknesses of the GCD test:
  1. It does not exploit knowledge about the loop bounds.
  2. Most often the gcd is one.
- The first weakness means the GCD Test might be unable to prove independence despite the solution actually lies outside the index space of the loop.
- The second weakness means independence usually cannot be proved.

GCD Test for Nested Loops and Multdimensional Arrays

- The GCD Test can be extended to cover nested loops and multidimensional arrays.
- The solution is then a vector and it usually contains unknowns.
- The Fourier-Motzkin Test described shortly takes the solution vector from this GCD Test and checks whether the solution lies within the loop bounds.
- Next we will look at unimodular matrices and Fourier elimination used by the Fourier-Motzkin Test.

Unimodular Matrices

- An integer square matrix A is unimodular if its determinant $det(\mathrm{A}) = \pm 1$.
- If A and B are unimodular, then A^{-1} exists and is itself unimodular, and $\mathrm{A} \times \mathrm{B}$ is unimodular.
- $\mathcal{I}$ is the $m \times m$ identity matrix.

Elementary Row Operations

- The operations
  - *reversal*: multiply a row by -1,
  - *interchange*: interchange two rows, and
  - *skewing*: add an integer multiple of one row to another row,

  are called the elementary row operations.
- With each elementary row operation, there is a corresponding *elementary matrix*.

Performing Elementary Row Operations

- To perform an elementary row operation on a matrix A, we can premultiply it with the corresponding elementary matrix.
- Assume we wish to interchange rows 1 and 3 in a 3×3 matrix A. The resulting matrix is formed by

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \times A.$$

- The elementary matrices are all unimodular.

3×3 Reversal Matrices

- $$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}.$$

3×3 Interchange Matrices

- $$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

3×3 Upper Skewing Matrices

- $$\begin{pmatrix} 1 & z & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

- $$\begin{pmatrix} 1 & 0 & z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

  and

- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & z \\ 0 & 0 & 1 \end{pmatrix}.$$

3×3 Lower Skewing Matrices

- $$\begin{pmatrix} 1 & 0 & 0 \\ z & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ z & 0 & 1 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & z & 1 \end{pmatrix}.$$

Echelon Matrices

- Let l_i denote the column number of the first nonzero element of matrix row i.
- A given $m \times n$ matrix A, is an *echelon matrix* if the following are satisfied for some integer ρ in $0 \leq \rho \leq m$:
  - rows 1 through ρ are nonzero rows,
  - rows $\rho + 1$ through m are zero rows,
  - for $1 \leq i \leq \rho$, each element in column l_i below row i is zero, and
  - $l_1 < l_2 < ... < l_\rho$.
- Informally: diagonal element i must only have zeroes below it
- Which of the following is not an echelon matrix?

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 0 & 0 & 7 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & 4 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{pmatrix}$$

Echelon reduction: essentially Gauss elimination

- Given an $m \times n$ matrix A, Echelon reduction finds two matrices U and S such that $\mathrm{U} \times \mathrm{A} = \mathrm{S}$, where U is unimodular and S is echelon.
- U remains unimodular since we only apply elementary row operations.

```
function echelon_reduce(A)
        U ← I_m
        S ← A
        i_0 ← 0
        for (j ← 1; j ≤ n; j ← j + 1) {
            if (there is a nonzero s_ij with i_0 < i ≤ m) {
                i_0 ← i_0 + 1
                i = m
                while (i ≥ i_0 + 1) {
                    while (s_ij ≠ 0) {
                        σ ← sign(s_(i−1)j × s_ij)
                        z ← ⌊|s_(i−1)j| / |s_ij|⌋
                        subtract σz(row i) from (row i − 1) in (U; S)
                        interchange rows i and i − 1 in (U; S)
                    }
                    i ← i − 1
                }
            }
        }
    return U and S
end
```

Example Echelon Reduction

- We will now show how one can echelon reduce the following matrix:

$$A = \begin{pmatrix} 2 & 2 \\ -3 & 1 \\ 3 & 1 \\ -4 & -2 \end{pmatrix}.$$

- We start with with $U = I_4$ and $S = A$ which we write as:

$$(U; S) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 0 & 3 & 1 \\ 0 & 0 & 0 & 1 & -4 & -2 \end{array}\right).$$

- Then we will eliminate the nonzero elements in S starting with $s_{41}, s_{31}, s_{21}, s_{42}$ and so on.

Example Echelon Reduction

- $j = 1, i_0 = 1, i = 4$. We always wish to eliminate s_{ij}, which currently means s_{41}.
- $\sigma \leftarrow -1$ and $z \leftarrow 0$. Nothing is subtracted from row 3.
- Then rows 3 and 4 are interchanged in $(U; S)$, resulting in:

$$(U; S) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 0 & 1 & -4 & -2 \\ 0 & 0 & 1 & 0 & 3 & 1 \end{array}\right).$$

Example Echelon Reduction

- We continue the inner while loop and find that $\sigma \leftarrow -1$ and $z \leftarrow 1$. Then $-1\times$ row 4 is subtracted from row 3, resulting in:

$$(\mathsf{U}; \mathsf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 1 & 0 & 3 & 1 \end{array}\right).$$

- Then rows 3 and 4 are interchanged, resulting in:

$$(\mathsf{U}; \mathsf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 0 & 3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \end{array}\right).$$

Example Echelon Reduction

- s_{41} is still zero, and the inner while loop is continued and $\sigma \leftarrow -1$ and $z \leftarrow 3$. Then $-3\times$ row 4 is subtracted from row 3:

$$(U; S) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 0 & 0 & 1 & 1 & -1 & -1 \end{array}\right).$$

- Then rows 3 and 4 are interchanged, resulting in:

$$(U; S) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Now the first $_{ij}$ has become zero and i is decremented.

Example Echelon Reduction

- $j = 1, i_0 = 1, i = 3$. We now wish to eliminate s_{31}. $\sigma \leftarrow +1$ and $z \leftarrow 3$. Then $3\times$ row 3 is subtracted from row 2:

$$(U; S) = \left(\begin{array}{rrrr|rr} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Then rows 2 and 3 are interchanged, resulting in:

$$(U; S) = \left(\begin{array}{rrrr|rr} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

Example Echelon Reduction

- $j = 1, i_0 = 1, i = 2$. We now wish to eliminate s_{21}. $\sigma \leftarrow -1$ and $z \leftarrow 2$. Then $-2\times$ row 2 is subtracted from row 1:

$$(U; S) = \left(\begin{array}{rrrr|rr} 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Interchanging rows 2 and 1 results in:

$$(U; S) = \left(\begin{array}{rrrr|rr} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

Example Echelon Reduction

- $j = 2, i_0 = 2, i = 4$. We now wish to eliminate s_{42}. $\sigma \leftarrow -1$ and $z \leftarrow 2$. $-2\times$ row 4 is subtracted from row 3:

$$(U; S) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Interchanging rows 4 and 3 results in:

$$(U; S) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

Example Echelon Reduction

- $j = 2, i_0 = 2, i = 3$. We now wish to eliminate s_{32}. $\sigma \leftarrow 0$ and $z \leftarrow 0$. Nothing is subtracted from row 2 but rows 3 and 2 are interchanged:

$$(U; S) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

At this point S is an echelon matrix and the algorithm stops (the outer while loop since $i = i_0$). As will turn out to be convenient later, we prefer positive values of s_{11} and therefore multiply with -1 finally resulting in:

$$(U; S) = \left(\begin{array}{cccc|cc} 0 & 0 & -1 & -1 & 1 & 1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

Recall from previous slides in this lecture

```
for (i = 0; i < 100; i += 1)
        for (j = 2*i + 4; j < i + 40; j += 1)
                a[2i-3j-1][2i+j-3] = f(a[-3i+4j+1][-i+2j+7]);
```

- $A = \begin{pmatrix} 2 & 2 \\ -3 & 1 \end{pmatrix}$ and $a_0 = (-1, -3)$.
  $B = \begin{pmatrix} -3 & -1 \\ 4 & 2 \end{pmatrix}$ and $b_0 = (1, 7)$.
- We want to find integer solutions to:

$$(i; j) \begin{pmatrix} A \\ -B \end{pmatrix} = b_0 - a_0.$$

- But better if we can prove none exist!

Solving a dependence equation

- Two references for the same variable: a matrix with n dimensions
- $m/2$ for-loops m loop index variables (`i,j,k` etc for each reference)
- That is: the loop index variables $i_1, i_2, \ldots, i_{m/2}$

$$\mathrm{xA} = \mathrm{c}$$

- x is an $1 \times m$ integer matrix
- A is an $m \times n$ integer matrix
- c is an $1 \times n$ integer matrix
- We find U and S such that UA = S.
- Then try to solve tS = c
- If there is solution, then: c = tS = tUA.
- So x = tU

An example

- Consider xA = c with

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \end{pmatrix} \begin{pmatrix} 2 & 2 \\ -3 & 1 \\ 3 & 1 \\ -4 & -2 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix}$$

- Firstly we use echelon reduction to find the matrices U and S.
- Then we solve tS = c

$$\begin{pmatrix} t_1 & t_2 & t_3 & t_4 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & -2 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix}$$

We find that t = $(2, -1, t_3, t_4)$, where t_3 and t_4 are arbitrary integers.

- We then find x:

$$\mathsf{x} = \mathsf{tU} = \begin{pmatrix} 2 & -1 & t_3 & t_4 \end{pmatrix} \begin{pmatrix} 0 & 0 & -1 & -1 \\ 0 & 0 & 4 & 3 \\ 1 & 0 & 2 & 2 \\ 0 & 1 & 5 & 3 \end{pmatrix} =$$

$$(t_3, t_4, 2t_3 + 5t_4 - 7, 2t_3 + 3t_4 - 5)$$

Fourier Elimination

- Suppose we find an integer solution x to xA = c.
- The next question is if the solution is within the loop bounds.
- Unfortunately, the problem of solving a linear integer inequality is NP-complete.
- Instead the compiler looks for a rational solution and only if no rational solution within the loop bounds exists, it ignores that pair of array references.

Fourier Elimination

- In 1827 Fourier published a method for solving linear inequalities in the real case.
- This is sometimes called Fourier-Motzkin elimination
- Utpal Banerjee, a leading compiler researcher at Intel has written a very good book series about parallelization calls it Fourier's method of elimination.

Fourier Elimination

- An interesting question is how frequently Fourier elimination finds a real solution when there is no integer solution. Some special cases can be exploited.
- For instance, if a variable x_i must satisfy $2.2 \leq x_i \leq 2.8$ then there is no integer solution.
  Otherwise, if we find eg that $2.2 \leq x_i \leq 4.8$ then we may try the two cases of setting $x_i = 3$ and $x_i = 4$, and see if there still is a real solution.
- It is easiest to understand Fourier elimination if we first look at an example.

Fourier Elimination

- Assume we wish to solve the following system of linear inequalities.

$$\begin{array}{rcrcr} 2x_1 & - & 11x_2 & \leq & 3 \\ -3x_1 & + & 2x_2 & \leq & -5 \\ x_1 & + & 3x_2 & \leq & 4 \\ -2x_1 & & & \leq & -3 \end{array}$$

- We will first eliminate x_2 from the system, and then check whether the remaining inequalities can be satisfied. To eliminate x_2, we start out with sorting the rows with respect to the coefficients of x_2:

$$\begin{array}{rcrcr} -3x_1 & + & 2x_2 & \leq & -5 \\ x_1 & + & 3x_2 & \leq & 4 \\ 2x_1 & - & 11x_2 & \leq & 3 \\ -2x_1 & & & \leq & -3 \end{array}$$

Fourier Elimination

- First we want to have rows with positive coefficients of x_2, then negative, and lastly zero coefficients.
- Next we divide each row by its coefficient (if it is nonzero) of x_2:

$$
\begin{array}{rcrcr}
\frac{-3}{2}x_1 & + & x_2 & \leq & \frac{-5}{2} \\
\frac{1}{3}x_1 & + & x_2 & \leq & \frac{4}{3} \\
\frac{2}{11}x_1 & - & x_2 & \geq & \frac{3}{11}
\end{array}
$$

Of course, the $\leq$ becomes $\geq$ when dividing with a negative coefficient. We can now rearrange the system to isolate x_2:

$$
\begin{array}{rcrcccrcr}
 & & & & x_2 & \leq & \frac{3}{2}x_1 & - & \frac{5}{2} \\
 & & & & x_2 & \leq & -\frac{1}{3}x_1 & + & \frac{4}{3} \\
\frac{2}{11}x_1 & - & \frac{3}{11} & \leq & x_2 & & & &
\end{array}
$$

Fourier Elimination

- At this point, we make a record of the minimum and maximum values that x_2 can have, expressed as functions of x_1. We have:

$$b_2(x_1) \leq x_2 \leq B_2(x_1)$$

where

$$\begin{aligned} b_2(x_1) &= \frac{2}{11}x_1 \\ B_2(x_1) &= \min\left(\frac{3}{2}x_1 - \frac{5}{2}, -\frac{1}{3}x_1 + \frac{4}{3}\right) \end{aligned}$$

Fourier Elimination

- To eliminate x_2 from the system, we simply combine the inequalities which had positive coefficients of x_2 with those which had negative coefficients (ie, one with positive coefficient is combined with one with negative coefficient):

$$\begin{array}{rcrcrcr} \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & \frac{3}{2}x_1 & - & \frac{5}{2} \\ \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & -\frac{1}{3}x_1 & + & \frac{4}{3} \end{array}$$

- These are simplified and the inequality with the zero coefficient of x_2 is brought back:

$$\begin{array}{rcr} -\frac{29}{22}x_1 & \leq & -\frac{49}{22} \\ -\frac{17}{33}x_1 & \leq & \frac{53}{33} \\ -2x_1 & \leq & -3 \end{array}$$

Fourier Elimination

- We can now repeat parts of the procedure above:

$$
\begin{array}{rcr}
x_1 & \leq & \frac{53}{17} \\
x_1 & \geq & \frac{49}{29} \\
x_1 & \geq & \frac{3}{2}
\end{array}
$$

- We find that

$$
\begin{array}{rcr}
b_1() & = & \max(49/29, 3/2) = 49/29 \\
B_1() & = & 53/17
\end{array}
$$

The solution to the system is $\frac{49}{29} \leq x_1 \leq \frac{53}{17}$ and $b_2(x_1) \leq B_2(x_1)$ for each value of x_1.

Fourier Elimination

```
procedure fourier_motzkin_elimination (x, A, c)
        r ← m,    s ← n,    T ← A,    q ← c
        while (1) {
            n_1 ← number of inqualities with positive t_rj
            n_2 ← n_1 + number of inqualities with negative t_rj
            Sort the inequalities so that the n_1 with t_rj > 0 come first,
                then the n_2 − n_1 with t_rj < 0 come next,
                and the ones with t_rj = 0 come last.
            for (i = 1; i ≤ r − 1; i ← i + 1)
                for (j = 1; i ≤ n_2; j ← j + 1)
                    t_ij ← t_ij / t_rj
            for (j = 1; i ≤ n_2; j ← j + 1)
                q_j ← q_j / t_rj
            if (n_2 > n_1)
                b_r(x_1, x_2, ..., x_{r−1}) = max_{n_1+1≤j≤n_2} (− Σ_{i=1}^{r−1} t_ij x_i + q_i)
            else
                b_r ← −∞
            if (n_1 > 0)
                j_r(x_1, x_2, ..., x_{r−1}) = min_{n_1+1≤j≤n_2} (− Σ_{i=1}^{r−1} t_ij x_i + q_i)
            else
                B_r ← ∞
            if (r = 1)
                return make_solution()
```

Fourier Elimination

```
        /* We will now eliminate x_r. */
        s' ← s − n_2 + n_1(n_2 − n_1)
        if (s' = 0) {
            /* We have not discovered any inconsistency and */
            /* we have no more inequalities to check. */
            /* The system has a solution. */
            The solution set consists of all real vectors (x_1, x_2, ..., x_m),
            where x_{r−1}, x_{r−2}, ..., x_1 are chosen arbitrarily, and
            x_m, x_{m−1}, ..., x_r must satisfy
            b_i(x_1, x_2, ..., x_{i−1}) ≤ x_i ≤ B_i(x_1, x_2, ..., x_{i−1}) for r ≤ i ≤ m.
            return solution set.
        }
        /* There are now s' inequalities in r − 1 variables. */
        The new system of inequalities is made of two parts:
        ∑_i^{r−1} (t_ik − t_il) x_i ≤ q_k − q_j for 1 ≤ k ≤ n_1, n_1 + 1 ≤ j ≤ n_2
        ∑_i^{r−1} t_ij x_i ≤ q_j for n_2 + 1 ≤ j ≤ s
        and becomes by setting r = r ← 1 and s ← s':
        ∑_i^r t_ij x_i ≤ q_j for 1 ≤ j ≤ s
    } end


function make_solution()
    /* We have come to the last variable x_1. */
    if (b_1 > B_1 or (there is a q_j < 0 for n_2 + 1 ≤ j ≤ s))
        return there is no solution
    The solution set consists of all real vectors (x_1, x_2, ..., x_m),
        such that b_i(x_1, x_2, ..., x_m) ≤ x_i ≤ B_i(x_1, x_2, ..., x_m) for 1 ≤ i ≤ m.
    return solution set.
end
```

Summary

- In the case of a loop nest of height m and an n-dimensional array, we use the matrix representation of the references $\mathrm{iA} + \mathrm{a}_0 = \mathrm{jB} + \mathrm{b}_0$, or equivalently:

$$(\mathrm{i}; \mathrm{j}) \begin{pmatrix} \mathrm{A} \\ -\mathrm{B} \end{pmatrix} = \mathrm{b}_0 - \mathrm{a}_0,$$

  where the **A** and **B** have m rows and n columns.
- We find a $2m \times 2m$ unimodular matrix **U** and a $2m \times n$ echelon matrix **S** such that

$$\mathrm{U} \begin{pmatrix} \mathrm{A} \\ -\mathrm{B} \end{pmatrix} = \mathrm{S}.$$

- If there is a $2m$ vector **t** which satisfies $\mathrm{tS} = \mathrm{b}_0 - \mathrm{a}_0$ then the GCD test cannot exclude dependence, and if so...
- ..., the computed t will be input to the Fourier-Motzkin Test.

The Fourier-Motzkin Test

- If the GCD Test found a solution vector t to $tS = c$, these solutions will be tested to see if they are within the loop bounds.
- Recall we wrote

$$x = (i; j) \begin{pmatrix} A \\ -B \end{pmatrix} = b_0 - a_0.$$

- We find x from:

$$x = (i; j) = tU$$

- With U_1 being the left half of U and U_2 the right half we have:

$$i = tU_1$$
$$j = tU_2$$

- These should be used in the loop bounds constraints.

The Fourier Motzkin Test

- Recall the original loop bounds are:

$$\left.\begin{array}{rcccl} p_0 & \leq & IP & & \\ & & IQ & \leq & q_0 \end{array}\right\}$$

- The solution vector t must satisfy:

$$\left.\begin{array}{rcccl} p_0 & \leq & tU_1P & & \\ & & tU_1Q & \leq & q_0 \\ p_0 & \leq & tU_2P & & \\ & & tU_2Q & \leq & q_0 \end{array}\right\}$$

- If there is no integer solution to this system, there is no dependence.
- Recall, however, the system is solved with real or rational numbers so the Fourier-Motzkin Test may fail to exclude independence.

After Data Dependence Analysis

- When we have performed data dependence analysis of all pairs of references to the same arrays, we have a **dependence matrix**, denoted D.
- Some rows will be due to some array and other rows due to some other arrays.
- It's the dependence matrix that determines which transformations we can do.
- As mentioned, in the optimizing compilers course inner loop transformations are studied for SIMD vectorization and software pipelining.
- We will look at outer loop parallelization.

Unimodular Transformations

- A **unimodular transformation** is a loop transformation completely expressed as a unimodular matrix U.
- A loop nest L is changed to a new loop nest L_U with loop index variables:

$$\begin{aligned} K &= IU \\ I &= KU^{-1} \end{aligned}$$

- The same iterations are executed but in a different order.
- A new iteration order might make parallel execution possible.
- Before generating code for the new loop, the loop bounds for K must be computed from the original bounds:

$$\left.\begin{array}{cccc} p_0 & \leq & IP & \\ & & IQ & \leq & q_0 \end{array}\right\}$$

Computing the New Index Variables

- With

$$\left.\begin{array}{rcccc} p_0 & \leq & IP & & \\ & & IQ & \leq & q_0 \end{array}\right\}$$

$$I = KU^{-1}$$

We use Fourier elimination also to find the loop bounds from

$$\left.\begin{array}{rcccc} p_0 & \leq & KU^{-1}P & & \\ & & KU^{-1}Q & \leq & q_0 \end{array}\right\}$$

- The bounds are found starting with k_1, k_2 etc.
- This is the reason why we want to have an invertible transformation matrix.

New Array References

- All array references are rewritten to use the new index variables.
- Conceptually we could calculate, at the beginning of each loop iteration,
$$\mathsf{I} = \mathsf{K}\mathsf{U}^{-1}$$
and then use this vector I in the original references, on the form:
$$x[\mathsf{I}\mathsf{A} + \mathsf{a}_0]$$
- We don't do that of course and instead replace each reference with
$$x[\mathsf{K}\mathsf{U}^{-1}\mathsf{A} + \mathsf{a}_0]$$
- Here $\mathsf{K}\mathsf{U}^{-1}\mathsf{A} + \mathsf{a}_0$ can be calculated at compile-time.

The Distance Matrix

- The set of all vectors of dependence distances is represented by the **distance matrix** D.
- We are free to swap the rows of D since it really is a set of dependences.
- Unimodular transformations require that all dependences are uniform, i.e. with known constants.
- Consider a uniform dependence vector $d = j - i$.
- With index variables $K = I\ U$ we have $d_U = jU - iU = dU$.
- Therefore, given a dependence matrix D and a unimodular transformation U, the dependences in the new loop L_U become:

$$D_U = DU$$

Valid Distance Matrices

- The sign, **lexicographically**, of a vector is the sign of the first nonzero element.
- A distance vector can never be lexicographically negative since it would mean that some iteration would depend on a future iteration.
- Therefore no row in the new distance matrix $\mathrm{D_U} = \mathrm{DU}$ may be lexicographically negative.
- If we would discover a lexicographically negative row in $\mathrm{D_U}$, that loop transformation is invalid, such as the second row of the following $\mathrm{D_U}$:

$$D_U = \begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}$$

Outer Loop Parallelization

- By **outer loops** is meant all loops starting with the outermost loop.
- While we always can find a unimodular matrix through which we can parallelize the inner loops, this is not the case for outer loops.
- To parallelize the inner loops, we need to assure that all loop carried dependences are carried at the outermost loop.
- In other words, the leftmost column of the distance matrix D_U simply should consist only of positive numbers!
- For outer loop parallelization, D_U instead should have leading zero columns.

Rank of a Matrix

- A column of a matrix is linearly independent if it cannot be expressed as a linear combination of the other columns.
- The rank of a matrix is the number of linearly independent columns.
- For instance, an identity matrix $\mathrm{I_m}$ with m columns has $\mathrm{rank}(\mathrm{I_m}) = m$.
- Any unimodular $m \times m$-matrix U has $\mathrm{rank}(\mathrm{U}) = m$.
- A matrix with zero columns must have a rank less than the number of columns.
- So, since $\mathrm{D_U} = \mathrm{DU}$, if $\mathrm{D_U}$ should have a rank less than m, it must be D which contributes with that.

Outer Loop Parallelization Example

- Assume we have the distance matrix **D** defined as:

$$D = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix}$$

- With this distance matrix, only the innermost loop can be executed in parallel.
- We want a D_U with positive rows and zero columns to the left.
- For example:

$$D_U = \begin{pmatrix} 0 & ? & ? \\ 0 & ? & ? \\ 0 & ? & ? \end{pmatrix} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} U$$

- If $\mathit{rank}(D) = 3$ then such a U cannot exist.

Steps towards Finding U

- We start with transposing D:

$$\mathrm{D}^{\mathrm{t}} = \begin{pmatrix} 6 & 0 & 1 \\ 4 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix}$$

- Using the Echelon reduction algorithm, we compute:
  - a unimodular matrix V
  - an echelon matrix S
- Such that $\mathrm{VD}^{\mathrm{t}} = \mathrm{S}$, e.g.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -2 \\ 1 & -1 & -1 \end{pmatrix} \mathrm{D}^{\mathrm{t}} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 0 & 0 & 0 \end{pmatrix}$$

- If $rank(\mathrm{D}) = 3$ then the last row of S would have a non-zero element
- With n zero rows in S we can parallelize n outer loops

More Steps towards Finding U

- We have $\mathrm{VD^t} = \mathrm{S}$:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -2 \\ 1 & -1 & -1 \end{pmatrix} \begin{pmatrix} 6 & 0 & 1 \\ 4 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 0 & 0 & 0 \end{pmatrix}$$

- With $n = 1$ zero row in S we can parallelize $n = 1$ outer loop
- Then we find an $m \times (n+1)$ matrix A such that DA has n zero columns and then a column with elements greater than zero.
- This A will be used to find U.
- We saw that multiplying the last n rows of V with the columns of $\mathrm{D^t}$ produced the n zero rows in S.
- The first n columns of A should be the last n rows of V, for $n = 1$:

$$\mathrm{DA} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & ? \\ -1 & ? \\ -1 & ? \end{pmatrix} = \begin{pmatrix} 0 & ? \\ 0 & ? \\ 0 & ? \end{pmatrix}$$

Finding the Rest of A

- Finding the last column of A is easy. Denote it u.

$$\mathrm{DA} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & u_1 \\ -1 & u_2 \\ -1 & u_3 \end{pmatrix} = \begin{pmatrix} 0 & \geq 1 \\ 0 & \geq 1 \\ 0 & \geq 1 \end{pmatrix}$$

- Multiplying each row of D with u should produce a positive number:

$$\begin{array}{ccccccc} 6u_1 & + & 4u_2 & + & 2u_3 & \geq & 1 \\ & & u_2 & - & u_3 & \geq & 1 \\ u_1 & & & + & u_3 & \geq & 1 \end{array}$$

- We find u to be e.g. $\mathrm{u} = (1, 1, 0)$.

$$\mathrm{A} = \begin{pmatrix} 1 & 1 \\ -1 & 1 \\ -1 & 0 \end{pmatrix}$$

Computing U

- Given A, using a variant of echelon reduction, we find a unimodular matrix U such that $A = UT$

$$A = \begin{pmatrix} 1 & 1 \\ -1 & 1 \\ -1 & 0 \end{pmatrix} = UT = \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Computing L_U

- With this loop transformation matrix U, we get the following new dependence matrix D_U:
  $D_U = DU$
- i.e.

$$D_U = \begin{pmatrix} 0 & 10 & 6 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = DU = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

- The compiler does not actually need to compute D_U but it is a nice internal check to verify no row is lexicographically negative.
- The new loop L_U is constructed as explained before:
- A loop nest L is changed to a new loop nest L_U with loop index variables:

$$K = IU$$

- New array references and new loop bounds must be computed.
- We have already seen both of these two, but repeat them for convenience on the next two slides.

Recall: Computing the New Index Variables

- With

$$\left.\begin{array}{rcl} p_0 & \leq & \mathrm{I}P \\ & & \mathrm{I}Q \leq q_0 \end{array}\right\}$$

$$\mathrm{I} = KU^{-1}$$

We use Fourier elimination to find the loop bounds from

$$\left.\begin{array}{rcl} p_0 & \leq & KU^{-1}P \\ & & KU^{-1}Q \leq q_0 \end{array}\right\}$$

- The bounds are found starting with k_1, k_2 etc.

Recall: New Array References

- All array references are rewritten to use the new index variables.
- Conceptually we could calculate, at the beginning of each loop iteration,

$$I = KU^{-1}$$

  and then use this vector I in the original references, on the form:

$$x[IA + a_0]$$

- We don't do that of course and instead replace each reference with

$$x[KU^{-1}A + a_0]$$

- Here $KU^{-1}A + a_0$ can be calculated at compile-time.

Summary

- Using linear algebra it is sometimes possible to automatically parallelize for-loops
- Optimizing compilers rewrite loops with while or gotos to for-loops when possible
- All these transformations can be expressed in a matrix which is then used to generate a new loop (this belongs to the category of elegant computer science).